Buildsysteme

Buildsysteme werden in der Softwareentwicklung eingesetzt. Sie führen automatisiert alle Schritte durch, die nach der Implementierung zur Erzeugung der fertigen, lauffähigen Software durchgeführt werden müssen. Darunter fallen Schritte wie Parsen und Kompilieren des Codes, aber auch das Erzeugen von Verzeichnisstrukturen, das Einbinden externer Dateien oder das Erzeugen einer Dokumentation.

Für eine bessere Strukturierung der auszuführenden Schritte werden diese zu Zielen zusammengefasst. Ein Ziel kann die Ausführung mehrerer Schritte umfassen.

Zwischen den Zielen bestehen Abhängigkeiten. Einige Ziele können nur erreicht werden, wenn zuvor andere Ziele erreicht wurden. So kann z. B. Quellcode erst kompiliert werden, wenn Bibliotheksklassen an die richtige Stelle kopiert wurden. Aus diesem Grund kennt jedes Ziel eines Projekts die Ziele, von denen die eigene Ausführbarkeit direkt abhängt.

Aufgaben

1 Material 1 zeigt einen Ausschnitt aus dem UML-Klassendiagramm eines Buildsystems.

1.1 Erläutern Sie die im Diagramm dargestellten Beziehungen.

(3 BE)

1.2 Erklären Sie das Fachkonzept „Polymorphie“ am Beispiel der Klassen *Schritt*, *DateiKompilieren* und *VerzeichnisAnlegen*. Gehen Sie dabei auch auf die Methode *ausführen()* in den drei Klassen ein.

(3 BE)

2 Material 2 zeigt die Ziele eines Beispielprojektes mit ihren Laufzeiten. Für jedes Ziel werden in dem Feld *abhängigVon* diejenigen Ziele gespeichert, von denen die eigene Ausführbarkeit direkt abhängt. In Material 2 ist diese Abhängigkeit mithilfe von Pfeilen dargestellt. Der Pfeil von Ziel A zu Ziel B bedeutet, dass Ziel A erst erledigt werden kann, wenn Ziel B erreicht wurde.

2.1 Begründen Sie anhand zweier verschiedener Aspekte, dass es sich bei dem Diagramm in Material 2 nicht um einen Baum handelt.

(2 BE)

2.2 Obwohl es sich nicht um einen Baum handelt, ist es sinnvoll, von der Höhe eines Ziels zu sprechen. Entwickeln Sie eine sinnvolle Definition für die Höhe eines Ziels und ermitteln Sie die Höhe des Ziels C.

(2 BE)

2.3 Implementieren Sie die Methode *ermittleErreichbarkeit()* der Klasse *Ziel*, die prüft, ob alle Ziele, von denen das Ziel direkt abhängt, bereits erreicht wurden. In diesem Fall gibt die Methode *true* zurück, sonst *false*.

(4 BE)

3 Die Klasse *Buildsystem* (Material 1) bietet mit Hilfe der Methode *erledigeZiele(Initialziele: Ziel[])* die Möglichkeit, ein oder mehrere Ziele des Projektes zu erledigen. Diese Ziele werden hierbei als Initialziele bezeichnet.
Das Buildsystem ermittelt zunächst alle Ziele, die zusätzlich erledigt werden müssen, um die Initialziele erledigen zu können. Diese Ziele werden zusammen mit den Initialzielen in einer einfach verketteten Liste gesammelt. Dabei soll jedes Ziel in der Liste nur einmal vorkommen.
Hinweis: Gehen Sie zur Vereinfachung davon aus, dass zu Beginn des beschriebenen Prozesses keines der Ziele des Projektes erreicht wurde.

3.1 Geben Sie alle Ziele an, die erledigt werden müssen, wenn das Buildsystem die Initialziele B und E aus Material 2 erledigen soll.

(1 BE)

3.2 Material 3 zeigt das UML-Klassendiagramm der Klasse *Liste*. Modellieren Sie im Material 3 eine Ergänzung um die Klasse *Knoten* inklusive aller Attribute, Methoden und Beziehungen, so dass die Datenstruktur Objekte vom Typ *Ziel* verwalten kann.

(3 BE)

3.3 Implementieren Sie die Methode *enthält(einZiel: Ziel)* der Klasse *Liste*, die überprüft, ob die Liste bereits das Objekt *einZiel* enthält. In diesem Fall gibt sie *true* zurück, sonst *false*.

(4 BE)

3.4 Die Methode *erstelleZielliste(Initialziele: Ziel[]): Liste* der Klasse *Buildsystem* ruft für jedes Initialziel die rekursiv arbeitende Methode *fügeAbhängendeZieleHinzu(einZiel: Ziel, Zielliste: Liste)* auf, die zu *Zielliste* sowohl *einZiel* hinzufügt als auch alle Ziele, die zuvor erreicht worden sein müssen.

Implementieren Sie die Methode *fügeAbhängendeZieleHinzu(...)*.

(4 BE)

4 Das Abarbeiten der Zielliste aus Aufgabe 3 kann zeitlich sehr aufwändig sein. Ärgerlich ist es, wenn Ziele, die viel Laufzeit benötigen, zuerst abgearbeitet werden und anschließend Ziele mit kürzerer Laufzeit Fehler verursachen und der Buildvorgang abgebrochen werden muss. Da diese Fehler ihren Ursprung in vorausgegangenen Zielen haben, müssen Ziele erneut erledigt werden, die bereits erreicht waren.
Das Buildsystem überführt die Zielliste daher vor der Abarbeitung in eine Prioritätswarteschlange, die alle noch nicht abgearbeiteten Ziele in einem Feld *dieZiele* speichert. Vorne in der Prioritätswarteschlange befinden sich alle Ziele, die bereits erreichbar sind. Diese Ziele sind aufsteigend nach Laufzeit sortiert. Am Ende der Prioritätswarteschlange befinden sich noch nicht erreichbare Ziele in beliebiger Reihenfolge. Das Buildsystem entnimmt dieser Prioritätswarteschlange immer das erste Element und bearbeitet es. Der Quellcode der Klasse *Prioritätswarteschlange* ist in Material 4 gegeben.

4.1 Analysieren Sie die Methode *gibErstes()* der Klasse *Prioritätswarteschlange*.

(3 BE)

4.2 Zeichnen Sie die Belegung des Feldes *dieZiele*, die sich ergibt, nachdem der Konstruktor der Klasse *Prioritätswarteschlange* mit der Zielliste C, D, F, G, H (Material 2) aufgerufen wurde. Geben Sie auch an, welche Zahlenwerte die Attribute *Erstes*, *Letztes* und *LetztesErreichbares* nach Abarbeitung des Konstruktors enthalten.
Hinweis: Gehen Sie davon aus, dass keines der Ziele C – H bereits erreicht wurde.

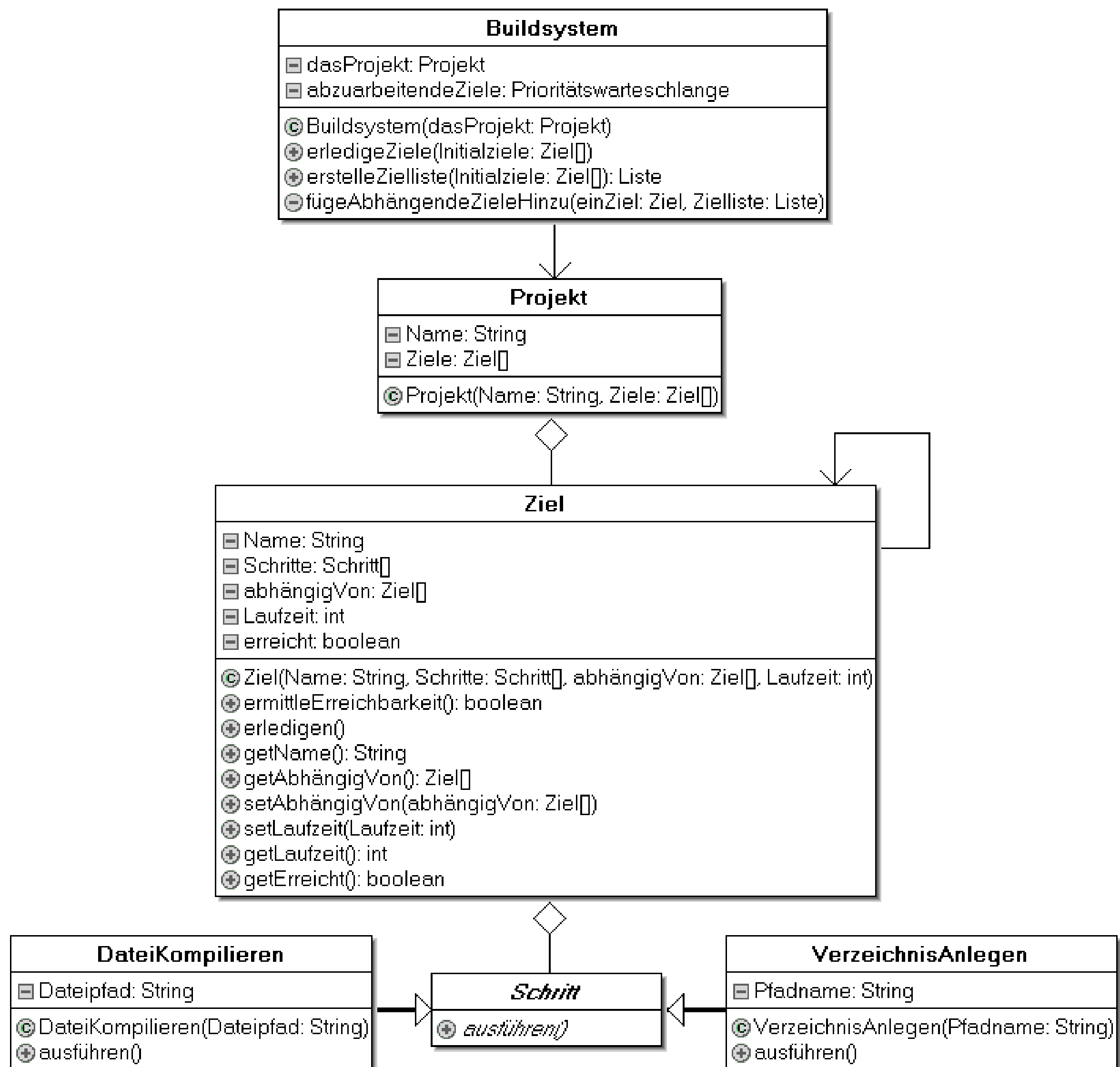
(3 BE)

4.3 Die Prioritätswarteschlange könnte die zu erledigenden Ziele alternativ in einer verketteten Liste speichern. Entscheiden Sie sich begründet für eine der beiden Datenstrukturen.

(3 BE)

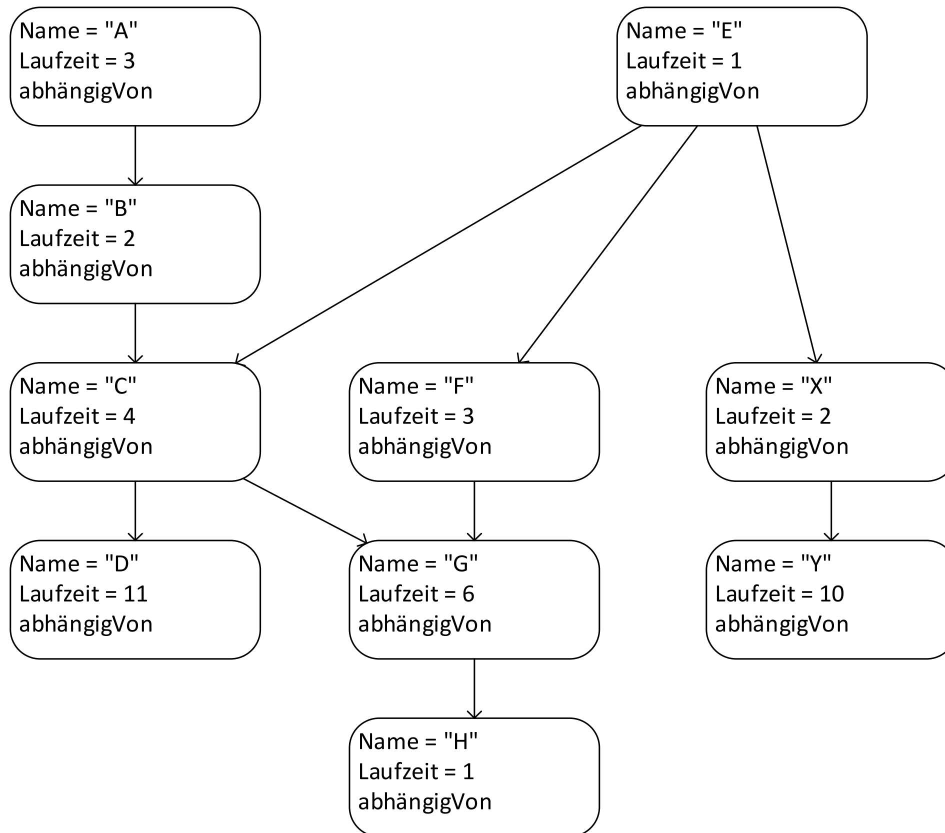
Material 1

Ausschnitt aus dem UML-Klassendiagramm eines Buildsystems



Material 2

Vereinfachtes Objektdiagramm von Zielen für ein Beispielprojekt



Material 3

Klassendiagramm der Klasse *Liste*



Material 4

Quellcode der Klasse Prioritätswarteschlange

```
public class Prioritätswarteschlange{
  private Ziel[] dieZiele;
  private int Erstes;
  private int LetztesErreichbares;
  private int Letztes;
  private int Anzahl;

  public Prioritätswarteschlange(Liste alleZiele){
    Anzahl = alleZiele.getAnzahl();
    dieZiele = new Ziel[Anzahl];
    Letztes = Anzahl - 1;
    int a = 0;
    int e = Letztes;
    Ziel temp;
    for(int i = 0; i <= Letztes; i++){
      temp = alleZiele.holeNächstes();
      if(temp.ermittleErreichbarkeit()){
        dieZiele[a] = temp;
        a++;
      } else {
        dieZiele[e] = temp;
        e--;
      }
    }
    LetztesErreichbares = e;

    for(int i = LetztesErreichbares; i > 0; i--){
      for(int j = 0; j < i; j++){
        if(dieZiele[j].getLaufzeit() > dieZiele[j + 1].getLaufzeit()){
          temp = dieZiele[j];
          dieZiele[j] = dieZiele[j+1];
          dieZiele[j+1] = temp;
        }
      }
    }
    Erstes = 0;
  }

  public Ziel gibErstes(){
    if(Anzahl == 0) return null;
    Ziel nächstesZiel = dieZiele[Erstes];
    Erstes = Erstes + 1;
    Anzahl = Anzahl - 1;
    return nächstesZiel;
  }

  public void umordnen(){
    /*Diese Methode muss jedes Mal aufgerufen werden,
    nachdem ein Ziel aus der Prioritätswarteschlange entfernt
    und erledigt worden ist, damit erreichbar gewordene Ziele
    mit der korrekten Priorisierung berücksichtigt werden.
    Die Implementierung der Methode ist hier aus
    Gründen der Übersichtlichkeit nicht angegeben*/
  }
}
```